LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-CONF-220714

Design and Implementation of Ceph: A Scalable Distributed File System

S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, C. Maltzahn

April 20, 2006

Symposium on Operation Systems Design and Implementation
Seattle, WA, United States
November 6, 2006 through November 8, 2006

Design and Implementation of Ceph: A Scalable Distributed File System

Sage A. Weil Scott A. Brandt Ethan L. Miller Darrell D. E. Long
Carlos Maltzahn

Abstract

File system designers continue to look to new architectures to improve scalability. Object-based storage diverges from server-based (*e. g.* NFS) and SAN-based storage systems by coupling processors and memory with disk drives, delegating low-level allocation to object storage devices (OSDs) and decoupling I/O (read/write) from metadata (file open/close) operations. Even recent object-based systems inherit decades-old architectural choices going back to early UNIX file systems, however, limiting their ability to effectively scale to hundreds of petabytes.

We present Ceph, a distributed file system that provides excellent performance and reliability with unprecedented scalability. Ceph maximizes the separation between data and metadata management by replacing allocation tables with a pseudo-random data distribution function (CRUSH) designed for heterogeneous and dynamic clusters of unreliable OSDs. We leverage OSD intelligence to distribute data replication, failure detection and recovery with semi-autonomous OSDs running a specialized local object storage file system (EBOFS). Finally, Ceph is built around a dynamic distributed metadata management cluster that provides extremely efficient metadata management that seamlessly adapts to a wide range of general purpose and scientific computing file system workloads.

We present performance measurements under a variety of workloads that show superior I/O performance and scalable metadata management (more than a quarter million metadata ops/sec).

1 Introduction

System designers have long sought to improve the performance of file systems, which have proved critical to the overall performance of an exceedingly broad class of applications. The scientific and high-performance computing communities in particular have driven advances in the performance and scalability of distributed storage systems, typically predicting more general purpose needs by a few years. Traditional solutions, exemplified by NFS [20], provide a straightforward model in which a server exports a file system hierarchy that clients can map into their local name space. Although widely used, the centralization inherent in the client/server model has proven a significant obstacle to scalable performance.

More recent distributed file systems have adopted architectures based on object-based storage, in which conventional hard disks are replaced with intelligent object storage devices (OSDs) which combine a CPU, network interface, and local cache with an underlying disk or RAID [25, 34, 9, 8, 37]. OSDs replace the traditional block-level interface with one in which clients can read or write byte ranges to much larger (and often variably sized) named objects, distributing low-level block allocation decisions to the devices themselves. Clients typically interact with a metadata server (MDS) to perform metadata operations (open, rename, etc.), while communicating directly with OSDs to perform file I/O (reads and writes), significantly improving overall scalability.

Existing systems adopting this model have continued to suffer from scalability limitations due to limited (or no) distribution of the metadata workload. Continued reliance on traditional file system principles like allocation lists and inode tables and a reluctance to delegate intelligence to the OSD have further limited scalability, performance, and increased the cost of reliability. We present Ceph, a distributed file system that provides both excellent performance and reliability with unparalleled scalability. Our architecture is based on the assumption that file systems at the peta- and exabyte scale are inherently dynamic: large systems are inevitably built incrementally, node failures are the norm rather than the exception, and the quality and character of workloads are constantly shifting over time.

Ceph seeks to decouple metadata from data operations

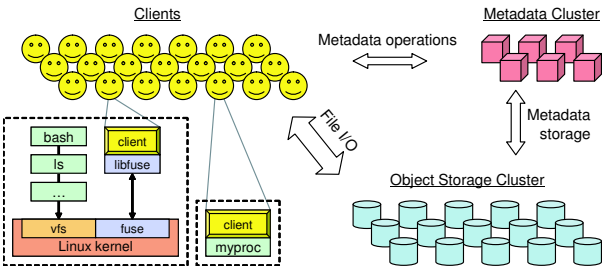

Figure 1: System architecture. Clients perform file I/O by communicating directly with OSDs. Each process can either link directly to a client instance, or interact with a mounted filesystem and share a client with other processes on the host.

by fully eliminating file allocation tables and replacing them with object name and distribution generating functions. This allows us to leverage the intelligence present in our OSDs to fully distribute the compexity surrounding data access, update serialization, replication and reliability, failure detection, and recovery operations. Finally, Ceph utilizes a dynamic distributed metadata cluster architecture that dramatically improves the scalability of metadata access, (and with it, the scalability of the entire system).

We discuss the goals and workload assumptions motivating our choices in the design of the architecture, analyze their impact on system scalability, performance, and reliability, and relate our experiences in implementing a functional system prototype.

2 System Overview

The Ceph file system has three main components: the client, each instance of which exposes a near-POSIX file system interfaces to a host and its applications; a cluster of OSDs, which collectively stores all file system data and metadata; and a metadata server cluster, which manages the file system namespace (file names and directories), while coordinating security, consistency and coherence issues (see Figure 1). We say that Ceph is nearly POSIX because we have found it appropriate to selectively both relax and extend the interface to improve system performance and better align the interface and consistency semantics with the needs of applications.

The primary goals of the architecture are scalability (to hundreds of petabytes and beyond), performance, and reliability. Scalability is considered in a variety of dimensions, including the overall storage capacity and throughput of the system, and performance in terms of individual clients, directories, or files. Our target workload may include such extreme cases as tens or hundreds of thousands of applications reading from or writing to the same file or creating files in the same directory. Such scenarios, common in scientific applications running on supercomputing clusters, are increasingly prescient of tomorrow's general purpose workloads. More importantly, we recognize that distributed file system workloads are inherently dynamic in nature, with significant variation in data and metadata access as active applications and data sets change over time. Ceph directly addresses the issue of scalability while simultaneously achieving high performance, reliability and availability through three fundamental design features.

Metadata and Data Decoupling First, Ceph maximizes the separation of file system metadata management from the storage of file data. Metadata operations (open, rename, etc.) are collectively managed by a metadata server (MDS) cluster, while clients interact directly with object storage devices (OSDs) to perform file I/O (reads and writes). Object-based storage has long promised to improve the scalability of file systems by delegating low-level block allocation decisions to individual devices. However, in contrast to existing object-based file systems [25, 34, 9, 8], which simply replace long per-file block lists with shorter object lists, Ceph eliminates allocation lists entirely. Instead, a simple function is used to name objects containing file data based on the inode number, byte range, and striping strategy, while a special-purpose data distribution function called CRUSH assigns objects to specific storage devices. This allows any party to simply calculate (instead of looking up) the name and location of objects comprising a file's contents, simplifying the design of the system and reducing the metadata cluster workload.

Dynamic Distributed Metadata Management Because file system metadata operations (such as *stat*, *readdir*, or *open*) make up as much as half of typical file system workloads [22], effective metadata management is critical to overall system performance. Ceph utilizes a novel metadata cluster architecture based on Dynamic Subtree Partitioning [32] that dynamically and intelligently distributes responsibility for managing the file system directory hierarchy among tens or even hundreds of metadata servers (MDSs). A coarse, hierarchical partition preserves locality in each MDS's workload, facilitating efficient updates and aggressive prefetching to improve performance for common workloads. Most significantly, the workload distribution among metadata servers is completely dynamic and based on current access patterns, allowing Ceph to most effectively utilize available MDS resources under any workload.

Reliable Distributed Object Storage Finally, Ceph delegates responsibility for data replication, failure detection, and failure recovery to the cluster of OSDs that is storing the data. In clusters composed of many thousands

of OSDs, device failure is frequent and expected. Moreover, large systems are dynamic: they are built incrementally, and grow or contract as new storage is deployed or old devices are decomissioned. All of these factors require that the distribution of data in the system evolve to effectively utilize available resources and maintain the desired level of data replication. Ceph's OSD cluster manages the details of replication, recovery, and data migration in a fully distributed fashion, while at a high level, Ceph clients and metadata servers treat the OSD cluster as collectively providing a single distributed and reliable object store. This approach allows Ceph to more effectively leverage the intelligence (CPU and memory) present on each OSD to achieve reliable, highly available object storage and self-healing with linear scaling.

The following sections describe the operation of the Ceph client, metadata server cluster, and distributed object store, and how they are affected by the critical features of our architecture. Where appropriate, we also describe the status of our prototype and how it differs from our design.

3 Client Operation

Each Ceph client provides a POSIX file system interface to applications. We begin by describing client operation both to introduce the overall operation and interaction of system components, and Ceph's interaction with applications that use it. Applications can link directly to the client code, which runs entirely in user space, either without modification by linking to a C library overloading POSIX file system calls (our prototype implements the most common, *e. g.* open, write, chdir), or by utilizing the native C++ class interface, which provides a slightly cleaner interface. A thin glue layer also allows a Ceph file system to be mounted natively under Linux via FUSE (a user space file system interface). The client maintains its own file data cache, independent of the kernel page or buffer caches, making it accessible to applications that link to the client directly.

3.1 File I/O and Capabilities

When a process opens a file, the client first sends a request to the MDS cluster. An MDS traverses the file system hierarchy to translate the file name into the *file inode*, which includes a unique inode number, the file owner, mode, size, and other per-file metadata. If the file exists and access is granted, the MDS returns the inode number, file size, and information about the striping strategy used to map file data into objects. The MDS may also issue the client node a *capability* for the file (if it does not already have one) specifying exactly which read or write operations it is authorized to perform. Capabilities issued to clients currently include four bits, controlling the client's ability to read, cache reads, write, or buffer writes. In the future, capabilities will also include security keys allowing clients to prove to OSDs that they are authorized to read or write data (the prototype trusts all clients). After a file has been opened, MDS involvement in file I/O is limited only to managing capabilities to preserve file consistency and achieve proper semantics.

Ceph generalizes a broad class of file data layout strategies to map file data onto a sequence of objects. Successive *stripe size* blocks of the file are assigned to the first *stripe width* objects, until objects reach a maximum *object size*, at which point a new set of objects is used. Whatever the layout (by default we simply break files into 1 MB chunks), an additional variable specifies how many replicas are stored of each object. To avoid any need for file allocation metadata, object names are constructed by concatenating the file inode number and the object number. Object replicas are then assigned to OSDs using CRUSH, a globally known mapping function (described in detail in Section 5.1).

For example, if one or more clients open a file for read-only access, an MDS grants them the capability to read and cache file content. Armed with the inode number, layout, and file size, the clients can locate all objects containing file data and read directly from the OSD cluster. Any objects or byte ranges that don't exist are defined to be file "holes", or zeros. Similarly, if a single client opens a file for writing, it is granted the capability to write with buffering, any data it generates at any offset in the file is simply written to the appropriate object on the appropriate OSD. When the file is closed, the client relinquishes the write capability and provides the MDS with the new file size (the largest offset written), which defines the set of objects that (may) exist and contain file data.

3.2 Client Synchronization

If a file is opened by multiple clients for writing, or for both reading and writing, things become more complex. POSIX semantics sensibly require that reads reflect any data previously written, and that writes are atomic (*e. g.* the result of overlapping, concurrent writes will reflect a particular order of occurrence). When a file is opened by multiple clients with either multiple writers or a mix of readers and writers, the MDS will revoke any previously issued read caching and write buffering capabilities, forcing all client I/O to be synchronous. That is, each application read or write operation will block until it is acknowledged by the OSD, effectively placing the burden of update serialization and synchronization with the OSD storing each object.[1]

Not surprisingly, synchronous I/O can be a perfor-

mance killer for applications (particularly those doing small reads or writes) due to the latency penalty (at least one round-trip to the OSD). Although read-write sharing is relatively rare in general-purpose workloads [22], it is more common in scientific computing applications [29], where performance is often critical. For this reason, it is often desireable to relax consistency at the expense of strict standards conformance in situations where applications do not rely on it. Although Ceph supports such relaxation via a global switch, and many other distributed file systems punt on this issue [20] (help! other examples besides nfs3? sorrento adopts a totally different consistency model, so it's no fair to rag on them here.), this is an imprecise and unsatisfying solution: either performance suffers, or strict consistency is lost.

For precisely this reason a set of high performance computing extensions to the POSIX I/O interface have been proposed by the HPC community. Most notably, these include an `O_LAZY` flag for *open* that allows applications to explicitly relax the usual coherency requirements to improve performance for a shared-write file [33]. Performance-conscious applications who manage their own consistency (*e. g.* by writing to different parts of the same file, a common pattern in HPC workloads [29]) are then allowed to buffer writes or cache reads when I/O would otherwise be performed synchronously. If desired, applications can then explicitly synchronize with two additional system calls: *lazyio_propogate* will flush a given file byte range to the appropriate OSDs, while *lazyio_synchronize* will ensure that the effects of previous propogations are reflected in any subsequent reads. The latter is implemented efficiently by provisionally invalidating cached data, such that subsequent read requests will be sent to the OSD but only return data if it is newer than what is in the cache. The Ceph synchronization model thus retains its simplicity by providing correct read-write and shared-write semantics between clients via synchronous I/O, and extending the application interface to relax consistency for performance conscious distributed applications.

3.3 Namespace Operations

Client interaction with the file system namespace is managed by the metadata server cluster. Both read operations (*e. g. readdir*, *stat*) and updates (*e. g. unlink*, *chmod*) are synchronously applied by the MDS to ensure serialization, consistency, correct security, and safety. For simplicity, no metadata locks or leases are issued to clients. For HPC workloads in particular, callbacks offer minimal upside at a high potential cost in complexity.

Instead, Ceph optimizes for the most common metadata access scenarios. A *readdir* followed by a *stat* of each file (*e. g.* `ls -l`) is an extremely common access pattern and notorious performance killer in large directories. A *readdir* in Ceph requires only a single MDS request, which fetches the entire directory list, including inode contents. By default, if a *readdir* is immediately followed by one or more *stat*s, the (briefly) cached information is returned; otherwise it is discarded. Although this relaxes coherence slightly in that an intervening inode modification may go unnoticed, we gladly make this trade for vastly improved performance. This behavior is explicitly captured by the *readdirplus* [33] extension, which returns *lstat* results directly with directory entries (as some OS-specific implementations of *getdir* already do).

Ceph can allow consistency to be further relaxed by caching metadata longer, much like earlier versions of NFS (which typically cache for 30 seconds). However, this approach breaks coherency in a way that is often critical to applications, such as those using *stat* to determine if a file has been updated—they either behave incorrectly, or end up waiting for old cached values to time out.

We opt instead to again provide correct behavior and extend the interface in instances where it adversely affects performance. This choice is most clearly illustrated by a *stat* operation on a file currently opened by multiple clients for writing. In order to return a correct file size and modification time, the MDS revokes any write capabilities on the file to momentarily stop updates and collect up-to-date size and mtime values from all writers. The highest values are returned with the *stat* reply, and capabilities are reissued to allow further progress. Although stopping multiple writers may seem drastic, it is necessary to ensure proper serializability. (For a single writer a correct up-to-date value can be retrieved from the writing client without interrupting progress.) Applications who find coherent behavior costly or unnecessary—victims of a POSIX interface that doesn't align with their needs—can opt to use the new *statlite* operation [33], which takes a bit mask specifying which inode fields are not required to be coherent.

4 Dynamically Distributed Metadata

shorten this way down, referencing details in sc04 paper! 1-2 pgs.

Metadata operations often make up as much as 50% of file system workloads [22], making the MDS cluster critical to overall system performance. Metadata management also presents a critical scaling challenge in distributed file systems: although capacity and aggregate I/O rates can scale almost arbitrarily with the addition of more storage devices, metadata operations involve a greater degree of interdependence that makes scalable consistency and coherence management more difficult.

Although metadata (like data) are ultimately stored on

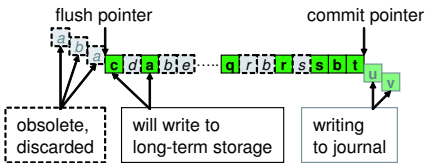

Figure 2: Each MDS maintains a very large journal (hundreds of megabytes) to efficiently stream commits to disk and to absorb repetitive updates.

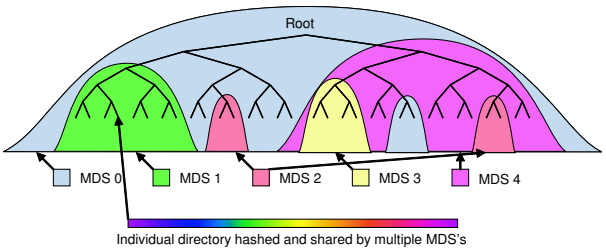

Figure 3: Ceph dynamically maps subtrees of the directory hierarchy to metadata servers based on current workloads. Individual directories are hashed across multiple nodes only when they become hot spots.

disk in a cluster of OSDs for safety, the MDS cluster maintains a large distributed in-memory cache to maximize performance. File and directory metadata in Ceph is very small, consisting only of directory entries (file names) and inode structs (80 bytes) with occasional file attributes. Unlike conventional file systems, no file allocation metadata is necessary—object names are constructed using the inode number, and distributed to OSDs using CRUSH. This simplifies the metadata workload and allows our MDS to efficiently manage a very large working set of files, independent of average file sizes. Our design further seeks to maximize locality and cache efficiency for each MDS, minimizing metadata related disk I/O through the use of a two-tiered storage strategy and efficient Dynamic Subtree Partitioning [32].

4.1 Metadata Storage

Although the MDS cluster aims to satisfy most metadata requests from its in-memory cache, all metadata updates must also be committed to disk for safety. A set of large, bounded, lazily flushed journals allows each MDS to quickly stream its updated metadata to disk in an efficient and distributed manner. The large per-MDS journals, each many hundreds of megabytes, also serve to absorb repetitive metadata updates (common to most workloads) such that when old journal entries are eventually flushed to long-term storage, many will have already been rendered obsolete, such that far fewer updates are required (see Figure 2). Although recovery is not yet implemented by our prototype, the journals are designed such that in the event of an MDS failure, another node can quickly rescan the journal to both partially reconstruct the contents of the failed node's in-memory cache (for quicker startup) and (in doing so) recover the Ceph file system state.

This two-tiered strategy provides the best of both worlds: streaming updates to disk in an efficient (sequential) fashion, and a vastly reduced re-write workload, allowing the long-term on-disk storage layout to be optimized for future read access. In particular, inodes are embedded directly within directories, allowing the MDS to prefetch entire directories with a single OSD read request, exploiting the high degree of directory locality present in most workloads [22]. Each directory's content, which consists of file names and inodes, is written to an OSD cluster using the same object striping and distribution strategy as file data. Inode numbers are managed with journaled updates and distributed free lists (or simply considered immutable, as in our prototype), while an auxiliary *anchor table* [30] is used to keep the rare inode with multiple hard links globally addressable by inode number—all without encumbering the overwhelmingly common case of singly-linked files with an enormous, sparsely populated and cumbersome conventional inode table.

4.2 Dynamic Subtree Partitioning

The primary-copy caching strategy makes a single authoritative MDS responsible for managing cache coherence and serializing and committing updates for any given piece of metadata. While most existing distributed file systems employ some form of static subtree-based partitioning to delegate this authority (usually forcing an administrator to carve their dataset into smaller static "volumes"), some recent and experimental file systems have tried hash functions to distribute directory and file metadata [25], effectively sacrificing locality for load distribution. Both approaches have critical limitations: static subtree partitioning fails to cope with dynamic workloads and data sets, while hashing destroys metadata locality and critical opportunities for efficient MDS prefetching and storage.

Ceph's metadata server cluster is based on a dynamic subtree partitioning strategy that allows it to adaptively distribute cached metadata hierarchically across a set of MDS nodes [32], as illustrated in Figure 3. Each MDS monitors the popularity of metadata within the directory hierarchy using counters with an exponential time decay attached to in-memory inodes. Any operation in-

crements the counter not only on the affected inode, but also on all of its ancestors up to the root directory, providing each MDS with a weighted tree describing the recent load distribution. Periodically MDS load values are compared across the MDS cluster, and arbitrary and variably-sized subtrees of the directory hierarchy are seamlessly reassigned and migrated to keep the workload evenly distributed. The combination of shared long-term storage and carefully constructed namespace locks allows such migrations to proceed by effectively transfering the appropriate contents of the in-memory cache (including "dirty" metadata that hasn't been flushed from the MDS journal) to the new subtree authority, with minimal impact on coherence locks or client capabilities tied to the individual metadata being migrated. The resulting subtree-based partition it kept coarse to minimize prefix replication overhead and preserve locality.

When metadata is replicated across multiple MDS nodes, inode contents are separated into three groups, each with different consistency semantics: security (owner, mode), file (size, mtime), and immutable (inode number, ctime, layout). While immutable fields never change, security and file locks are governed by independent finite state machines, each with a different set of states and transitions designed to accomodate different access and update patterns while minimizing lock contention. For example, owner and mode are required for the security check during path traversal but rarely change, requiring only a few states, while the file lock reflects a wider range of client access modes as it controls an MDS's ability to issue client capabilities or reply to *stat*.

4.3 Traffic Control

The distribution of the directory hierarchy across multiple nodes can balance a broad range of workloads, but does nothing to cope with hot spots or flash crowds, where many clients access the same directory or file. Ceph uses its knowledge of metadata popularity to provide a wide load distribution for hot spots only when it is needed and without incurring the associated overhead and loss of directory locality in the general case. The contents of heavily read directories (*e. g.* many opens) are selectively replicated across multiple nodes to distribute load. Directories that are particularly large or experiencing a heavy write workload (*e. g.* many file creations) are individually hashed (by file name) across the cluster, achieving a balanced distribution at the expense of directory locality. This dynamic approach allows Ceph to encompass a broad spectrum of partition granularities, dynamically capturing the benefits of both coarse and fine partitions in the specific regions where those strategies are most effective.

Every MDS response comes bundled with updated information about the authority and any replication of the relevant inode and its ancestors, which clients use to "learn" the metadata partition for the parts of the file system they interact with. Future metadata operations are directed either at the authority (for updates) or a random replica (for reads) based on the deepest known prefix of a given path. Normally clients learn the proper locations of unpopular (unreplicated) metadata and are able to contact the appropriate MDS directly. Clients accessing popular metadata, on the other hand, are told the metadata resides either on different or multiple MDS nodes, effectively bounding the number of clients believing any particular piece of metadata resides on any particular MDS, dispersing potential hot spots and flash crowds before they occur.

5 Distributed Object Storage

From a high level, Ceph clients and metadata servers view the object storage cluster (possibly tens or hundreds of thousands of OSDs) as a single logical object store with a single object namespace. Ceph's Reliable Autonomic Distributed Object Store (RADOS) achieves linear scaling in both capacity and aggregate performance by delegating management of object distribution replication, cluster expansion, and failure recovery to the OSDs in a fully distributed fashion.

5.1 Data Distribution with CRUSH

Ceph must distribute petabytes of data among an evolving cluster of thousands of storage devices such that device storage and bandwidth resources are most effectively utilized. In order to avoid imbalance (*e. g.* recently deployed devices mostly idle or empty) or load assymetries (*e. g.* new, hot data on new devices only), we adopt a distribution strategy that migrates a random subsample of existing data to new devices and uniformly redistributes data on removed devices. This stochastic approach is robust in that it performs equally well under any potential workload. Moreover, we utilize a function for data placement to avoid bottlenecks related to location metadata or allocation (*e. g.* block allocation in GPFS [yes? no?]).

Data distribution is accomplished by first mapping objects into *placement groups* (PGs), and then assigning each PG to be replicated across a list of OSDs. Each 128-bit object name (formed by combining the inode number and file object number) is translated into a 32-bit *placement key* via a simple hash function, the lowest pg_bits bits of which identify the object's placement group. Setting (or changing) pg_bits allows one to control the total number of placement groups as the system grows (or contracts) by potentially several orders of magnitude

over its lifetime. We currently choose a value that gives each OSD on the order of 100 PGs to limit the amount of replication-related metadata maintained by each OSD.

Placement groups are mapped to an ordered list of OSDs using the CRUSH algorithm (Controlled Replication Under Scalable Hashing) [31], a pseudo-random data distribution function that efficiently and robustly maps each PG to a list of OSDs upon which to store object replicas. This differs from conventional approaches (including other object-based file systems like Lustre and PanFS [25, 34]) in that data placement does not rely on any block or object lists on the MDS. In contrast, to locate any placement group (and hence any object) CRUSH requires only the placement group and an *OSD cluster map*: a compact, hierarchical description of the (weighted) devices comprising the storage cluster. This approach has two key advantages: first, it is completely distributed such that any party (client, OSD, or MDS) can independently calculate the location of any object; and second, the OSD cluster map is infrequently updated, virtually eliminating any distribution-related metadata that must be exchanged. In doing so, CRUSH simultaneously solves both the data distribution problem ("where should I store data") and the data location problem ("where did I store data").

The *OSD cluster map* hierarchy is structured to align with its physical composition and potential sources of failure. For instance, one might form a five-level hierarchy for an installation consisting of shelves full of OSDs, rack cabinets full of shelves, rows of cabinets, and many rows in each room. Each OSD is also assigned a weight value, which controls the relative amount of data it is assigned. CRUSH maps PGs onto OSDs based on *placement rules*, which define the level of replication and any constraints on placement. For example, one might specify that each placement group is to be replicated on three OSDs, all situated in the same row (to limit inter-row replication traffic) but separated into different cabinets (to minimize exposure to a power circuit or edge switch failure). The cluster map also includes a list of failed or inactive devices, and is tagged with a version number, which is incremented by an MDS each time the map changes. All OSD requests are tagged with the caller's map version, such that all parties agree on the current distribution of data, and map updates piggyback on OSD replies if the caller's map is found to be out of date.

5.2 Replication

Unlike systems like Lustre [25], which assume one can construct sufficiently reliable OSDs using mechanisms like RAID or fail-over on a SAN, we assume that in a multi-petabyte system OSD failure will the norm rather than the exception, and that at any point in time several OSDs are likely to be inoperable. To maintain system availability and ensure data safety in a scalable fashion, the Ceph OSD cluster manages its own replication of data, while taking steps to minimize the impact of replication on performance.

Data is replicated in terms of placement groups, each of which is assigned (by CRUSH) to an ordered list of n OSDs (for n-way replication). Clients send all write requests to the first non-failed OSD in an object's placement group (designated the *primary*), who applies the update locally, updates the version number associated with the object and PG, and forwards the update to any replica OSDs. When each replica applies the update and responds to the primary, the write is acknowledged to the client. This approach spares the client of any of the complexity surrounding synchronization or serialization between multiple object replicas, which can become onerous in the presence of other writers or failure and recovery operations. It also shifts the bandwidth consumed with replication activities from client to the OSD cluster's internal network, where we expect greater resources to be available. Object version numbers are based on a 64-bit Lamport-style clock maintained by the messaging layer, functioning like a timestamp; each PG's version is effectively defined as its most recently updated object's version.

5.3 Data Safety

In distributed storage systems, there are essentially two different reasons why data is written to shared storage. First, we are interested in making our updates visible to other clients. This should be quick: we'd like our applications to know their writes are visible as soon as possible, particularly when multiple writers or mixed readers and writers force clients to operate synchronously. Second, we are interested in data safety, such that we can know reliably that the data we've written is safely replicated and on disk and will survive power or other failures. Ceph's object store disassociates synchronization from safety when acknowledging updates, allowing the system to realize both low-latency updates for efficient application synchronization and well-defined data safety semantics.

Figure 4 illustrates the messages sent during an object update request. OSDs first apply the update to their in-memory buffer caches, and immediately reply with an *ack*. A final *commit* is only sent (perhaps many seconds) later when the update is safely committed to disk. We send the *ack* to the client only after the update is applied to the in-memory caches on all replicas, even though this increases client latency by an additional round-trip, because we want to seamlessly tolerate the failure of any single OSD. This leaves data exposed only to a simul-

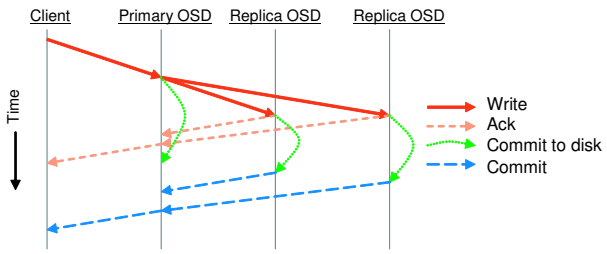

Figure 4: The OSD cluster responds with an *ack* after the write has been applied to the buffer caches on all OSDs replicating the object. Only after it has been safely committed to disk is a final *commit* notification sent to the client.

taneous power loss to all OSDs in the placement group, a risk we mitigate by separating replicas across different failure domains with the CRUSH distribution policy. The addition of client-driven ordered replay of uncommitted updates would protect against such coincident failures and allow us to acknowledge updates sooner, although this is not yet implemented by our prototype.

5.4 Failure Detection

The task of detecting OSD failures is partially distributed. For certain failures, such as disk hardware errors or corrupted data, OSDs can self-report. Failures that make an OSD unreachable on the network (*e. g.* power or network failure), however, require active monitoring. Ceph distributes OSD liveness verification by having each OSD ping a pseudo-random subset of its peers at regular intervals. In certain cases, existing inter-OSD replication traffic serves as a passive confirmation of liveness, with no additional communication overhead. If an OSD has not heard from a peer recently, an explicit ping is sent.

Because a wide variety of network anomalies may cause intermittent lapses in OSD connectivity, the MDS cluster collects failure reports and verifies failures to filter out transient or systemic problems (like a network partition) centrally. Updated OSD cluster maps are then broadcast to all OSDs[2]. This combination of distributed detection and a centralized coordinator in Ceph takes the best from both worlds: it allows fast detection without unduly burdening the MDS cluster, and resolves the occurrence of inconsistency with the arbitrament of the MDS cluster.

For greater flexibility, and in order to cope with intermitten OSD failures (*e. g.* a reboot), the Ceph OSD cluster considers two dimensions of OSD liveness: whether the OSD is reachable and responsive, and whether it is assigned data by CRUSH. A non-responsive OSD is initially marked as *down* in the OSD cluster map, indicating that it is not reachable. Any primary responsiblities (update serialization, replication) temporarily pass on to the next OSD for each of its placement groups. If after some interval the OSD does not recover, it is marked *out* of the data distribution map, at which point the acting primary becomes the new primary, another OSD is added to the placement group, and the new OSD initiates recovery by replicating all PG contents. Although our prototype does not yet cope with communication failures, in the future clients attempting to access a newly failed disk will simply time out and retry OSD operations, walking through all OSDs in the PG until the operations succeeds or a new OSD status (*failed* or *out*) is learned. Operations that are not directed at the acting primary are simply forwarded within the OSD cluster.

5.5 Recovery and Cluster Updates

more detail here.

The OSD cluster map will change due to OSD failures, recoveries, and explicit cluster changes such as the deployment of new storage. In all cases, a similar "recovery" procedure is followed. On boot, each OSD begins by retrieving the latest OSD cluster map from an MDS. It then iterates through all possible placement groups (as specified by *pg_bits*) and calculates the CRUSH mapping to discover which PGs it is responsible for, either as a primary or a replica. For every PG it replicates, the OSD establishes a peering session with the primary by sending the current version number for its placement groups. For each primary PG, the OSD collects the replica's versions and, if necessary, retrieves the full PG content list (object names and versions) in order to determine the correct (most recent) PG contents. This may involve waiting for PG OSDs from prior cluster map versions in order to locate all PG objects. At this point, each replica receives the latest PG content list, such that all parties agree to what the placement group contents *should* be, although their locally stored object set may not match. Each OSD is then independently responsible for retrieving missing or outdated objects from its peers. If an OSD receives a request involving a missing or outdated object, it delays processing and moves that object to the front of the recovery queue.

Whenever an OSD obtains an updated cluster map (either from an MDS or a peer), it quickly checks to see if any of its placement groups are affected (*e. g.* by an OSD newly marked as *failed*). The peering process (or an abbreviated version of it) is repeated for only those PGs where it is necessary (*i. e.* those that gained or lost an OSD). When an OSD crashes, is marked *failed*, and then recovers, for example, its designated role as primary is restored when it contacts the MDS on startup. When the

temporary acting primary receives the map update, it will realize it is no longer the primary and will re-peer with the recovered OSD. The recovered primary will realize it has an older version of the PG contents, and will retrieve the latest content list from the previous acting primary.

Because failure recovery and the resulting data re-replication is driven entirely by individual OSDs, each PG affected by a failed OSD will recover in parallel to (very likely) different replacement OSDs selected by CRUSH. This approach, based on the Fast Recovery Mechanism (FaRM) [39], decreases overall recovery times proportional to the number of PGs on each OSD, and improves overall data safety.

5.6 Object Storage with EBOFS

Although a variety of distributed file systems use existing kernel file systems like ext3 to manage local object storage on each OSD, we found their interface and performance to be poorly suited for object workloads. Each Ceph OSD manages its local object storage with EBOFS, our special-purpose Extent and B-tree based Object File System.

First, we found that the existing kernel file system interface limited our ability to explicitly understand when object updates were safely committed on disk. Although fully synchronous writes or full journaling could provide the desired safety, that ability comes with a heavy latency and performance penalty. The POSIX interface also fails to support atomic data and metadata (*e. g.* version attribute) update transactions, which were important for maintaining RADOS consistency. Implementing EBOFS entirely in user space and interacting directly with a raw block device allowed us to define our own low-level object storage interface and update semantics, which separate update serialization (for synchronization) from on-disk commits (for safety). EBOFS update functions return when the in-memory caches are updated, while providing asynchronous notification of commits via a callback interface.

A user space approach, aside from providing greater flexibility and easier implementation, also allowed us to avoid cumbersome interaction with the Linux VFS and page cache, both of which were designed for a different interface and workload. While most kernel file systems lazily flush updates to disk after some time interval, EBOFS aggressively schedules disk writes, and opts instead to cancel pending I/O operations when subsequent updates render them superfluous. This approach provides our low-level disk scheduler with larger I/O queues and the corresponding increase in I/O scheduling efficiency. It also affords us convenient access to the scheduler, making it simple to eventually prioritize workloads (*e. g.* client I/O versus recovery) or provide explicit quality of service guarantees [38].

Central to the EBOFS design is a robust, flexible yet fully integrated B-tree service that is used to locate objects on disk, manage block allocation, and index collections. Collections are named and efficiently indexed sets of objects, and are used by Ceph to group objects into placement groups. Block allocation is conducted in terms of extents—start and length pairs—instead of block lists, keeping metadata compact. Free block extents on disk are binned by size and stored in B-trees, sorted by location. This allows EBOFS to quickly and efficiently locate free space near the write position or related data on disk, while also limiting long-term fragmentation. With the exception of per-object block allocation information, all metadata is kept in memory for performance and simplicity (it is quite small, even for very large volumes).

Finally, EBOFS aggressively performs copy-on-write: data is only ever written to unallocated regions of disk. The single exception is a pair of versioned superblocks containing pointers to the B-tree root nodes (and by implication all EBOFS metadata), which are updated in an alternating pattern on each commit cycle (either on idle or every few seconds). On mount we simply choose the newest superblock, secure in the knowledge that each commit reflected a completely consistent and unadulterated snapshot of the local object file system state.

6 Performance and Scalability Evaluation

We evaluate our prototype under a range microbenchmarks to demonstrate key elements of the architecture and the scalability of its performance. In all of our tests, clients, OSDs, and MDSs are user processes running on a Linux cluster, communicating using TCP. In general, each OSD or MDS runs on its own host, while tens or hundreds of client instances may share the same host while generating workload.

6.1 Data Performance

Distribution of replication and failure recovery allows the aggregate performance of our OSD cluster to scale linearly with the size of the cluster. Our performance tests are conducted using 14 OSDs running on dual-processor Xeons with SCSI disks. We achieve perfect linear scaling until that point, after which throughput is limited by our network switch.

6.1.1 OSD Throughput

We begin by measuring the I/O performance of a 14-node cluster of OSDs. Figure 5 shows per-OSD throughput with varying write sizes (X) and levels of replication.

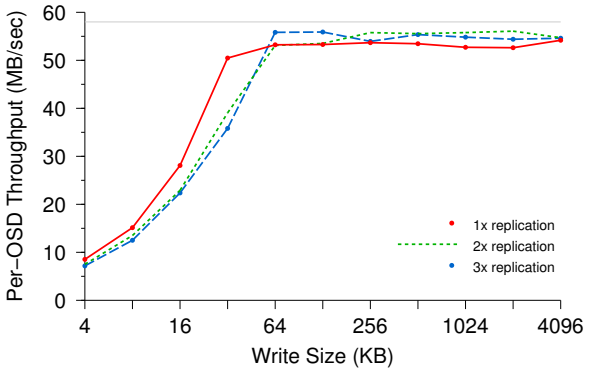

Figure 5: Per-OSD write performance. Data replication has minimal impact on OSD efficiency. If the number of OSDs is fixed, n-way replication reduces *effective* throughput by a factor of n because replicated data must be written to n OSDs. The horizontal line indicates the upper limit imposed by the physical disk.

The workload is generated by 400 client instances (on 20 additional nodes). Performance is ultimately limited by the raw disk bandwidth (around 50 MB/sec), shown by the dotted lines. Since replication doubles or triples disk I/O at a constant number of OSDs, we see correspondingly lower client data rates. Small writes are less efficient both because of greater communications overhead and coarse locking in the current EBOFS implementation.

Figures 6 and 7 compare per-OSD performance when general-purpose file systems (ext3, XFS, ReiserFS) are used for local object storage instead of EBOFS. Although small read and write performance in EBOFS currently suffers from coarse threading and locking, EBOFS very nearly saturates the available disk bandwidth for writes larger than 32 KB. EBOFS significantly outperforms the others for read workloads because data is laid out in large extents on disk when it is written in large increments.

6.1.2 Write Latency

Figure 8 shows the synchronous write latency for a single writer with varying write sizes (X) and levels of data replication. Because the primary OSD simultaneously retransmits updates to all replicas, for small writes we see a minimal latency increase for more than two replicas. For larger writes, the cost of retransmission begins to dominate; 1 MB writes (not shown) take 13 ms for one replica, and 3.6 times longer (47 ms) for four. Although we have considered the possibility of mitigating this effect by pipelining large writes to replicas, much like the Google File System [8], it is not clear that the payoff will

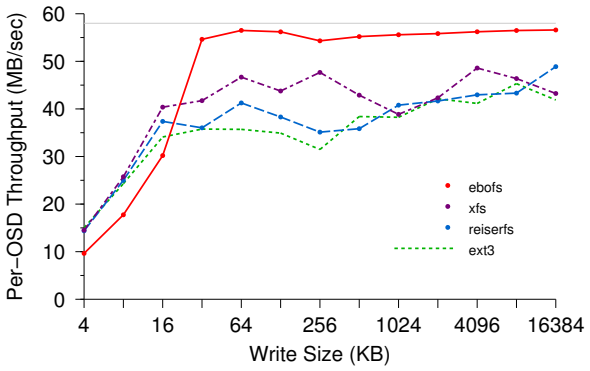

Figure 6: Write performance of EBOFS compared to general-purpose file systems. Although small writes suffer from coarse locking in our prototype, EBOFS very nearly saturates the disk for writes larger than 32 KB.

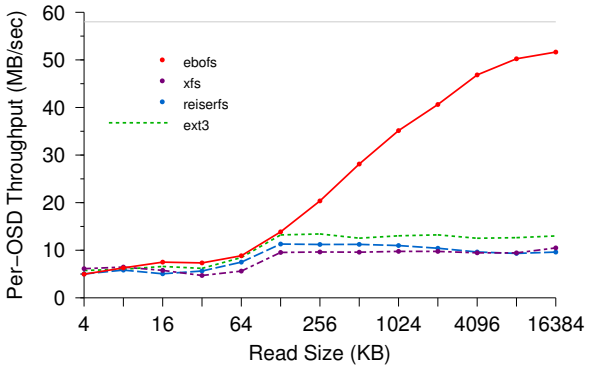

Figure 7: Read performance of EBOFS compared to general-purpose file systems. EBOFS lays out data in large extents when it is written in large increments, allowing it to offer significantly better performance than general purpose file systems.

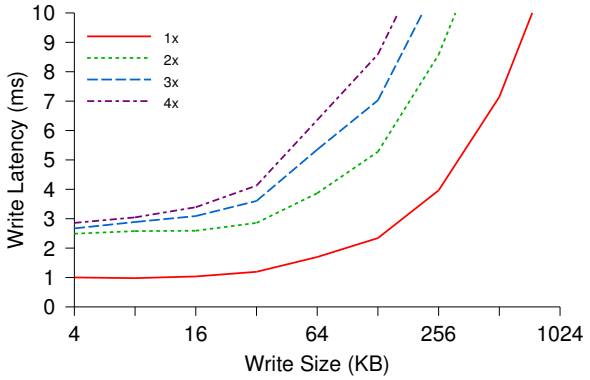

Figure 8: Write latency for varying write sizes and levels of replication. More than two replicas incurs minimal additional cost for small writes because replicated updates occur concurrently. For larger writes, transmission times dominate.

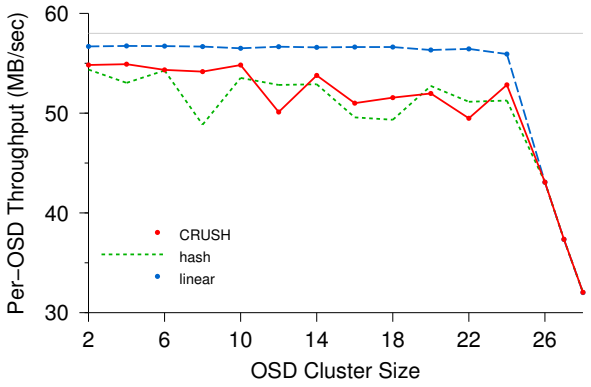

Figure 9: OSD write performance scales linearly with the size of the OSD cluster until we saturate our switch with 24 OSDs.

be very significant since it will only benefit synchronous large writes—write-sharing applications that are usually best off using `O_LAZY`. With consistency thus relaxed, clients can buffer small writes (*i. e.* zero latency seen by application) and submit only large, asynchronous writes to OSDs for maximum efficiency; the only latency seen by applications will be due to clients filling their caches and waiting for data to flush to disk. Although our prototype includes a simple buffer cache, we are in the process of redesigning it to better align with object layouts to accomodate data locks.

6.1.3 Data Distribution and Scalability

CRUSH distributes data to OSDs pseudo-randomly, such that OSD utilizations can be accurately modeled by a binomial or normal distribution—what one would expect from a perfectly random process. Relative variance in utilization decreases as the number of groups increases: for 100 placement groups the standard deviation is 10%, while for 1000 groups it drops to 3%. In our experiments the CRUSH distribution closely follows this model under a wide variety of cluster architectures, including varying device weights [31]. Because data placement is a stochastic function, devices can become overutilized with small probability, potentially dragging down performance; CRUSH allows individual OSD utilizations to be adjusted to correct such situations using a simple feedback mechanism.

Figure 9 shows per-OSD write throughput as cluster size increases, using both CRUSH, a simple hash function, and a linear striping strategy to distribute data. Throughput with the hash function and CRUSH drop slightly as OSD request queue lengths drift apart under our entangled client workload.

6.2 Metadata Performance

Ceph MDS performance is measured by generating a partial workload that does not include any data I/O. OSDs in these experiments are used solely for metadata journaling and storage.

6.2.1 Metadata Update Latency

We first consider the latency associated with synchronous metadata updates (*e. g. mknod* or *mkdir*) from clients. A single client creates a series of files and directories, which the MDS must synchronously journal to a cluster of OSDs for safety. We consider both diskless metadata servers, where all metadata is stored in a shared OSD cluster, and an architecture in which each MDS also has a local disk that serves as the primary OSD for its journal. Figure 10 shows the latency associated with metadata updates in both cases with varying levels of metadata replication (where zero corresponds to no journaling at all). Because the objects storing the metadata journal are first written to the primary OSD and then replicated to any additional OSDs, we see a jump in latency from zero to one and one to two replicas for the diskless MDS approach. When a local disk in each MDS acts as the primary, the initial hop from the MDS to the local OSD takes minimal time. This allows an MDS with a local disk to achieve update latencies for 2x replication similar to 1x in the diskless model. In both cases, more than two replicas incurs very little additional latency because replicated updates are conducted in parallel.

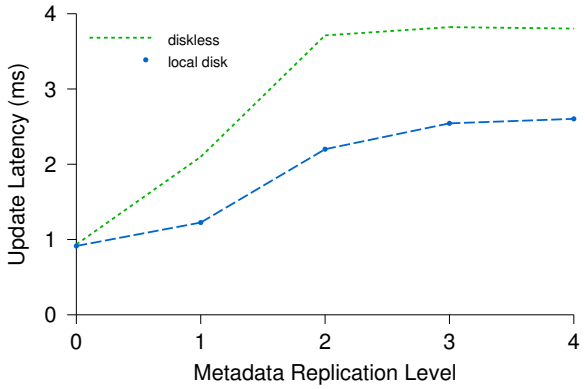

Figure 10: Metadata update latency for a diskless MDS and one with an OSD on the same host, with varying levels of replication. Zero corresponds to no journaling. We can lower the journaling latency with a local disk with similar reliability by avoiding the initial network round-trip.

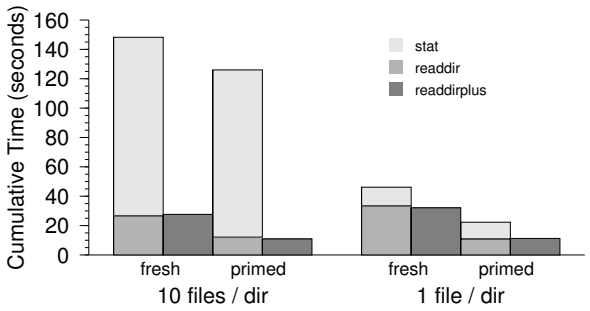

Figure 11: Cumulative time consumed by metadata operations during file system walk. A primed MDS cache eliminates an OSD access, while *readdirplus* or relaxed consistency eliminate MDS interaction for *stat*s following *readdir*.

6.2.2 Metadata Read Latency

The behavior of metadata reads (*e. g.* *readdir*, *stat*, file *open*) is more complex. We consider a client that walks a directory hierarchy by doing a *readdir* in each directory, doing a *stat* on each file, and then recursively descending into any subdirectories. Figure 11 shows cumulative time consumed by metadata operations during a walk over 10,000 nested directories. Ordinarily, cumulative *stat* times would dominate for larger directories. A primed MDS cache reduces *readdir* times by avoiding an OSD access. Subsequent *stat*s are not affected, because inode contents are embedded in directories, allowing the full directory contents to be fetched into the MDS cache with a single OSD access. Using *readdirplus*, which explicitly bundles *stat* and *readdir* results in a single MDS operation, or relaxing client consistency by allowing *stat*s immediately following a *readdir* to be served from client caches (the default), eliminates additional MDS interaction beyond the initial *readdir*.

6.2.3 Metadata Scaling

We evaluate the overall scalability of the metadata cluster by measuring MDS efficiency (per-MDS throughput) as the total size of the system increases. For each test, we use the same number of OSDs for metadata storage as there are MDS nodes, while an additional set of nodes each run between 25 and 50 independent instances of the client to generate the workload. Each workload consists of metadata operations only—no data I/O—allowing us to simulate the behavior of installations with tens of thousands of OSDs or more. These experiments were conducted on a 430 node partition of the `alc` Linux cluster at Lawrence Livermore National Labs.

Figure 12 shows the average per-MDS throughput (Y) as a function of MDS cluster size (X) for each workload we tested, such that perfect linear scaling would result in a horizontal line ($X\dot{Y}$ for total throughput). The *makedirs* workload has each client create a tree of nested directories four levels deep, with ten files and subdirectories in each directory. Average MDS throughput drops from 2000 ops per second with a small cluster, to just over 1000 ops per MDS per second (50% efficiency) with 128 metadata servers. In the *writefiles* workload, each client creates a thousands of files in the same (shared) directory. When the high write levels are detected, Ceph hashes the shared directory to distribute the workload across all MDS nodes. The *openshared* workload demonstrates read sharing by having each client repeatedly open and close ten shared files. The *openssh* workloads, each client replays a captured file system trace of a compilation of openssh in a private directory. One variant uses a shared `/lib` for moderate sharing, while the other shares `/usr/include`, which is very heavily read. The *openshared* and *openssh+include* workloads have the heaviest read sharing and show the worst scaling behavior, we believe due to poor replica selection by clients. *openssh+lib* scales better than the trivially separable *makedirs* because it contains relatively few metadata modifications and little sharing. Although we believe that contention in the network or threading of the communications further lowered performance for larger MDS clusters, our limited time with dedicated access to the larger cluster prevented a thorough investigation.

Ceph offers significantly greater scalability and efficiency than existing distributed file systems by several

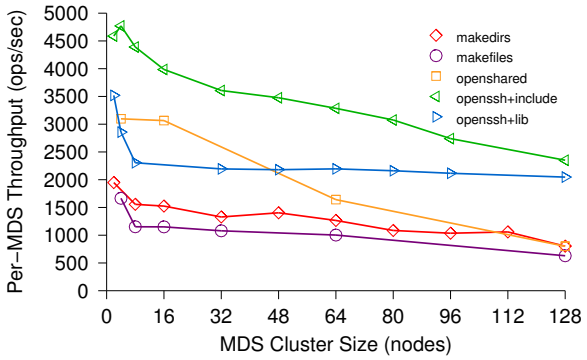

Figure 12: Per-MDS throughput under a variety of workloads and cluster sizes. As the cluster grows to 128 nodes, efficiency drops only slightly below perfectly linear (horizontal) scaling, allowing vastly improved performance over existing systems.

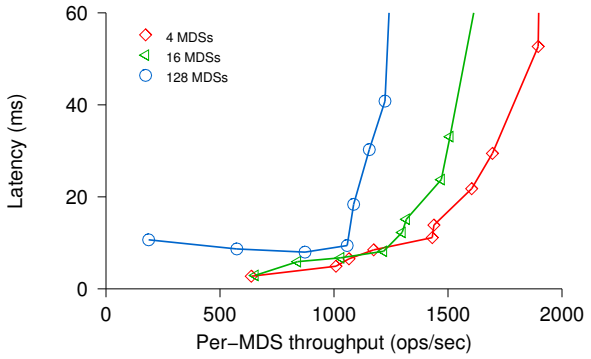

Figure 13: Average latency versus average per-MDS throughput for different cluster sizes. Larger clusters have lower average per-MDS throughput.

orders of magnitude, despite sub-linear scaling. A 128-nodes MDS cluster running our current prototype can service more than a quarter million metadata operations per second (128 nodes at 2000 ops/sec). Because metadata transactions are independent of data I/O and metadata size is independent of file size, this corresponds to installations with potentially many hundreds of petabytes of storage, depending on average file size. For example, scientific applications creating checkpoints on LLNL's Bluegene/L might involve 64 thousand nodes with two processors each writing to a file in the same directory (as in the *makefiles* workload). While BGL's current storage system peaks at 6,000 metadata ops/sec and would many minutes to complete each checkpoint, a 128-node Ceph MDS cluster would finish in two seconds. If each file were only 10 MB (quite small by HPC standards) and OSDs sustain 50 MB/sec, such a cluster could write 1.25 TB/sec, saturating at least 25,000 OSDs (50,000 with replication). If each OSD were 250 GB, such a system would store more than six petabytes. More importantly, Ceph's dynamic metadata distribution approach allows an MDS cluster (of any size) to reallocate MDS resources based on the current workload, even when all clients access metadata previously assigned to a single MDS, making it significantly more versatile and adaptable than any static partitioning strategy.

Figure 13 plots latency versus per-MDS throughput for a 4-, 16-, and 64-node MDS cluster under the *makedirs* workload. Larger clusters have imperfect load distributions, resulting in lower average per-MDS throughput (but, of course, much higher total throughput) and slightly higher latencies.

7 Experiences

Over the course of designing the major subsystems in Ceph and integrating them into a single system, we ran into a few surprises, both in terms of elements that were easier and more difficult that anticipated.

7.1 Metadata Decoupling and Smart OSDs

We were pleasantly surprised to discover the extent to which replacing file allocation metadata with a globally known distribution function (originally, RUSH [12]) became a simplifying force in our design. Although this placed greater demands on the distribution function itself, once we realized exactly what those requirements were (in terms of scalability, flexibility, and reliability), CRUSH was able to deliver. In return, we vastly simplified our metadata workload and on-disk layout, and provided our OSDs with complete and independent knowledge of the data distribution. The latter enabled us to delegate nearly all responsibility for data replication, migration, failure detection, and recovery to the OSDs, distributing these mechanisms in a way that effectively leveraged their bundled CPU and memory. RADOS has also opened the door to a range of additional features that elegently map onto our OSD model, such as bit error detection (as in the Google File System [8]), dynamic replication of data based on workload (simliar to AutoRAID [36]), and scalable OSD-managed object locks for improved write-sharing performance.

7.2 Local Object Storage

Although it was tempting to leverage existing kernel file systems for local object storage (as many other systems have done [25, 11, 8]), we recognized early on that a file system tailored for object workloads could offer better

performance [29]. What we did not anticipate was the disparity between the existing kernel (POSIX) file system interface and our requirements, which became evident as we developed the RADOS replication and reliability mechanisms. Our user-space implementation of EBOFS was surprisingly quick to develop, offered very satisfying performance, and exposed an interface perfectly suited to our requirements.

7.3 Metadata Load Balancing

One of the largest lessons in Ceph was the importance of the MDS load balancer to overall scalability, and the complexity of effectively choosing what metadata to migrate where and when. Although in principle our design and goals seem quite simple, the reality of distributing an evolving workload among over a hundred MDSs highlighted additional subtleties. Most notably, it becomes increasingly clear that MDS performance has a wide range of performance bounds, including CPU utilization, memory (and cache efficiency), and network or I/O limitations, any of which may be limiting performance at any point in time. Furthermore, it is difficult to quantitatively capture the balance between total throughput and fairness; under certain circumstances imbalanced metadata distributions can increase overall throughput [32].

7.4 Implementation Surprises

Finally, implementation of the client file system interface posed a greater challenge than anticipated. Although the use of FUSE vastly simplified the implementation task by eliminating the need for any kernel work, it introduced its own set of idiosyncrasies. `DIRECT_IO` disabled the kernel page cast but didn't support *mmap*, forcing us to modify FUSE to invalidate clean pages as a workaround. FUSE's use of the Linux VFS inode cache and its insistence on performing its own security checks proved annoying as it results in many unnecessary *getattr*s (*stat*s) for even simple application calls. Finally, page-based I/O between kernel and user space limits overall I/O rates. Although linking directly to client avoids FUSE issues, overloading system calls in user space introduces a new set of issues (most of which we have yet to fully examine). An in-kernel client is likely inevitable.

8 Related Work

High-performance, scalable file systems have long been a goal of the high-performance computing (HPC) community. HPC systems place a heavy load on the file system [19, 26, 29], placing a high demand on the file system to prevent it from becoming a bottleneck. As a result, there have been many scalable file systems that attempt to meet this need; however, these file systems do not support the same level of scalability that Ceph does. Some large-scale file systems, such as OceanStore [13] and Farsite [1] are designed to provide petabytes of highly reliable storage, and may be able to provide simultaneous access to thousands of separate files to thousands of clients. However, these file systems are not optimized to provide high-performance access to a small set of files by tens of thousands of cooperating clients. Bottlenecks in subsystems such as name lookup prevent these systems from meeting the needs of a HPC system. Similarly, grid-based file systems such as LegionFS [35] are designed to coordinate wide-area access and are not optimized for high performance in the local file system.

StorageTank [16]

Parallel file and storage systems such as Vesta [7], Galley [18], RAMA [17], PVFS and PVFS2 [6, 14], the Global File System [27] and Swift [5] have extensive support for striping data across multiple disks to achieve very high data transfer rates, but do not have strong support for scalable metadata access or robust data distribution for high reliability. For example, Vesta permits applications to lay their data out on disk, and allows independent access to file data on each disk without reference to shared metadata. However, Vesta, like many other parallel file systems, does not provide scalable support for metadata lookup. As a result, these file systems typically provide poor performance on workloads that access many small files as well as workloads that require many metadata operations. They also typically suffer from block allocation issues: blocks are either allocated centrally or, in the Global File System, via a lock-based mechanism. As a result, these file systems do not scale well to write requests from thousands of clients to thousands of disks. Similarly, the Google File System [8] is optimized for very large files and a workload consisting largely of reads and file appends, and is not well-suited for a more general HPC workload because it does not support high-concurrency general purpose access to the file system.

Recently, many file systems and platforms, including Federated Array of Bricks (FAB) [23], Kybos [37], Lustre [3, 25], GPFS [24], the Panasas file system [34], pNFS [11], Sorrento [28], and zFS [21] have been designed around network-attached storage [9, 10] or the closely related object-based storage paradigm [2]. All of these file systems can stripe data across network-attached devices to achieve very high performance, but they do not have the combination of scalable metadata performance, expandable storage, fault tolerance, and POSIX compatibility that Ceph provides. pNFS [11] and the Panasas object-based file system [34] stripe data across network-attached disks to deliver very high data transfer rates, but

they both suffer from a bottleneck in metadata lookups. Lustre [3, 25] has similar functionality: it supports nearly arbitrary striping of data across object storage targets, but it hashes path names to metadata servers. This approach distributes the metadata load, but destroys locality and makes POSIX compatibility difficult, despite schemes such as LH3 [4]. GPFS [24] also suffers from metadata scaling difficulties; while block allocation is largely lock-free, as it is in most object-based storage systems, metadata is not evenly distributed, causing congestion in metadata lookups. Moreover, none of these systems permits a client to locate a particular block of a file without consulting a centralized table. Sorrento [28] alleviates this problem somewhat and evenly distributes data and metadata among all of the servers, but only performs well in environments with low levels of write sharing in which processors work on disjoint data sets. FAB [23] focuses on continuously providing highly reliable storage; while its performance is acceptable, FAB provides very high reliability at the cost of somewhat reduced performance.

9 Future Work

A variety of core Ceph elements have not yet been implemented. These include the MDS failure detection and recovery, and several remaining POSIX calls on the client.

Although the Ceph dynamically replicates metadata when flash crowds access single directories or files, the same is not yet true of file data. We plan to allow OSDs to dynamically adjust the level of replication for individual objects based on workload, and to distribute read traffic across all OSDs in the (potentially expanded) placement group when appropriate. This will allow scalable access to small amounts of data, and may facilitate fine-grained load balancing using a mechanism similar to D-SPTF [15].

We plan to experiment with per-object shared or exclusive locks, managed and issued by OSDs, as a means of avoiding strictly synchronous I/O in shared-write or read-write situations. This will likely involve client-driven access prediction in order to request the appropriate locks before data is accessed by the application.

We are working on developing a quality of service architecture to allow both aggregate class-based traffic prioritization and OSD-managed reservation based bandwidth and latency guarantees. This work will likely combine the flexible EBOFS disk scheduler with previous OSD disk bandwidth shaping research [38], providing an infrastructure that will help balance RADOS replication and recovery operations with regular workload. A number of other EBOFS enhancements are planned, including an update journal to reduce the commit latency. We plan to distribute journal entries among several preallocated regions spread across the disk to minimize the impact on low-level disk scheduling. Data scouring, checksums, or other bit-error detection mechanisms will also improve overall data safety.

Finally, we plan on further investigating the practicality and potential benefits of client callbacks on namespace to inode translation metadata. For static regions of the file system, this would allow opens (for read) to occur without MDS interaction (*e. g.* compiler interaction with `/usr/include`).

10 Conclusions

Ceph addresses three critical challeneges of storage systems—scalability, performance, and reliability—by occupying a unique point in the design space. By shedding design assumptions (like allocation lists) found in nearly all existing systems, we maximally separate data from metadata management, allowing them to scale independently. This separation relies on CRUSH, a data distribution function that generates a stochastic distribution (much like a hash function) allowing clients to calculate (instead of looking up) object locations. CRUSH enforces data replica separation across failure domains for improved data safety while efficiently coping with the inherently dynamic nature of large storage clusters, where devices failures, expansion and cluster restructuring are the norm.

We leverage intelligent OSDs to manage data replication, failure detection and recovery, low-level disk allocation, scheduling, and data migration due to cluster expansion without encumbering any central server(s). Although objects can be considered files and stored by OSDs using a general-purpose file system, we found that EBOFS provides more appopriate semantics and superior performance by addressing the specific workloads and interface requirements present in Ceph.

Finally, Ceph's metadata management architecture addresses one of the most vexing problems in highly scalable storage—how to efficiently provide a single uniform directory hierarchy obeying POSIX directory semantics with performance that scales with the number of metadata servers. Ceph's dynamic subtree partitioning is a uniquely scalable approach, offering both efficiency and the ability to adapt to varying workloads.

11 Acknowledgments

This research was funded in part by the Lawrence Livermore National Laboratory. In particular, we would like to thank Bill Loewe, Tyce McLarty, Terry Heidelberg, and everyone else at Livermore who talked to us about their storage trials and tribulations, and who helped facilitate our two days of dedicated access time on `alc`. We would

also like to thank IBM for donating the 32-node cluster that aided in much of the OSD performance testing, and the NSF, who paid for the switch upgrade. Finally, we would like to thank the students and faculty of the SSRC for their input and support.

12 Availability

Ceph is licensed under the LGPL and is available at `http://ceph.sourceforge.net/`

References

[1] ADYA, A., BOLOSKY, W. J., CASTRO, M., CHAIKEN, R., CERMAK, G., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)* (Boston, MA, Dec. 2002), USENIX.

[2] AZAGURY, A., DREIZIN, V., FACTOR, M., HENIS, E., NAOR, D., RINETZKY, N., RODEH, O., SATRAN, J., TAVORY, A., AND YERUSHALMI, L. Towards an object store. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies* (Apr. 2003), pp. 165–176.

[3] BRAAM, P. J. The Lustre storage architecture. http://www.lustre.org/documentation.html, Cluster File Systems, Inc., Aug. 2004.

[4] BRANDT, S. A., XUE, L., MILLER, E. L., AND LONG, D. D. E. Efficient metadata management in large distributed file systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies* (Apr. 2003), pp. 290–298.

[5] CABRERA, L.-F., AND LONG, D. D. E. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems 4*, 4 (1991), 405–436.

[6] CARNS, P. H., LIGON, W. B., ROSS, R. B., AND THAKUR, R. PVFS: a parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference* (Atlanta, GA, Oct. 2000), pp. 317–327.

[7] CORBETT, P. F., AND FEITELSON, D. G. The Vesta parallel file system. *ACM Transactions on Computer Systems 14*, 3 (1996), 225–264.

[8] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)* (Bolton Landing, NY, Oct. 2003), ACM.

[9] GIBSON, G. A., NAGLE, D. F., AMIRI, K., BUTLER, J., CHANG, F. W., GOBIOFF, H., HARDIN, C., RIEDEL, E., ROCHBERG, D., AND ZELENKA, J. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (San Jose, CA, Oct. 1998), pp. 92–103.

[10] GIBSON, G. A., AND VAN METER, R. Network attached storage architecture. *Communications of the ACM 43*, 11 (2000), 37–45.

[11] HILDEBRAND, D., AND HONEYMAN, P. Exporting storage systems in a scalable manner with pNFS. Tech. Rep. CITI-05-1, CITI, University of Michigan, Feb. 2005.

[12] HONICKY, R. J., AND MILLER, E. L. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *Proceedings of the 18th International Parallel & Distributed Processing Symposium (IPDPS 2004)* (Santa Fe, NM, Apr. 2004), IEEE.

[13] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Cambridge, MA, Nov. 2000), ACM.

[14] LATHAM, R., MILLER, N., ROSS, R., AND CARNS, P. A next-generation parallel file system for Linux clusters. *LinuxWorld* (Jan. 2004), 56–59.

[15] LUMB, C. R., GANGER, G. R., AND GOLDING, R. D-SPTF: Decentralized request distribution in brick-based storage systems. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Boston, MA, 2004), pp. 37–47.

[16] MENON, J., PEASE, D. A., REES, R., DUYANOVICH, L., AND HILLSBERG, B. IBM Storage Tank—a heterogeneous scalable SAN file system. *IBM Systems Journal 42*, 2 (2003), 250–267.

[17] MILLER, E. L., AND KATZ, R. H. RAMA: An easy-to-use, high-performance parallel file system. *Parallel Computing 23*, 4 (1997), 419–446.

[18] NIEUWEJAAR, N., AND KOTZ, D. The Galley parallel file system. In *Proceedings of 10th ACM International Conference on Supercomputing* (Philadelphia, PA, 1996), ACM Press, pp. 374–381.

[19] NIEUWEJAAR, N., KOTZ, D., PURAKAYASTHA, A., ELLIS, C. S., AND BEST, M. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems 7*, 10 (Oct. 1996), 1075–1089.

[20] PAWLOWSKI, B., JUSZCZAK, C., STAUBACH, P., SMITH, C., LEBEL, D., AND HITZ, D. NFS version 3: Design and implementation. In *Proceedings of the Summer 1994 USENIX Technical Conference* (1994), pp. 137–151.

[21] RODEH, O., AND TEPERMAN, A. zFS—a scalable distributed file system using object disks. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies* (Apr. 2003), pp. 207–218.

[22] ROSELLI, D., LORCH, J., AND ANDERSON, T. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference* (San Diego, CA, June 2000), USENIX Association, pp. 41–54.

[23] SAITO, Y., FRØLUND, S., VEITCH, A., MERCHANT, A., AND SPENCE, S. FAB: Building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2004), pp. 48–58.

[24] SCHMUCK, F., AND HASKIN, R. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)* (Jan. 2002), USENIX, pp. 231–244.

[25] SCHWAN, P. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium* (July 2003).

[26] SMIRNI, E., AYDT, R. A., CHIEN, A. A., AND REED, D. A. I/O requirements of scientific applications: An evolutionary view. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing (HPDC)* (1996), IEEE, pp. 49–59.

[27] SOLTIS, S. R., RUWART, T. M., AND O'KEEFE, M. T. The Global File System. In *Proceedings of the 5th NASA Goddard Conference on Mass Storage Systems and Technologies* (College Park, MD, 1996), pp. 319–342.

[28] TANG, H., GULBEDEN, A., ZHOU, J., STRATHEARN, W., YANG, T., AND CHU, L. A self-organizing storage cluster for parallel data-intensive applications. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)* (Nov. 2004).

[29] WANG, F., XIN, Q., HONG, B., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MCLARTY, T. T. File system workload analysis for large scale scientific computing applications. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies* (College Park, MD, Apr. 2004), pp. 139–152.

[30] WEIL, S. A. Scalable archival data and metadata management in object-based file systems. Tech. Rep. SSRC-04-01, University of California, Santa Cruz, May 2004.

[31] WEIL, S. A., BRANDT, S. A., MILLER, E. L., AND MALTZAHN, C. CRUSH: Controlled, scalable, decentralized placement of replicated data. Tech. Rep. SSRC-06-01, University of California, Santa Cruz, Jan 2006.

[32] WEIL, S. A., POLLACK, K. T., BRANDT, S. A., AND MILLER, E. L. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)* (Pittsburgh, PA, Nov. 2004), ACM.

[33] WELCH, B. POSIX IO extensions for HPC. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)* (Dec. 2005).

[34] WELCH, B., AND GIBSON, G. Managing scalability in object storage systems for HPC Linux clusters. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies* (Apr. 2004), pp. 433–445.

[35] WHITE, B. S., WALKER, M., HUMPHREY, M., AND GRIMSHAW, A. S. LegionFS: A secure and scalable file system supporting cross-domain high-performance applications. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC '01)* (Denver, CO, 2001).

[36] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID hierarchical storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)* (Copper Mountain, CO, 1995), ACM Press, pp. 96–108.

[37] WONG, T. M., GOLDING, R. A., GLIDER, J. S., BOROWSKY, E., BECKER-SZENDY, R. A., FLEINER, C., KENCHAMMANA-HOSEKOTE, D. R., AND ZAKI, O. A. Kybos: self-management for distributed brick-base storage. Research Report RJ 10356, IBM Almaden Research Center, Aug. 2005.

[38] WU, J. C., AND BRANDT, S. A. Qos support in object-based storage devices. In *International Workshop on Storage Network Architecture and Parall I/Os (SNAPI '05)* (St. Louis, MO, Sept. 2005), pp. 41–48.

[39] XIN, Q., MILLER, E. L., AND SCHWARZ, T. J. E. Evaluation of distributed recovery in large-scale storage systems. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC)* (Honolulu, HI, June 2004), pp. 172–181.

Notes

[1] Achieving atomicity is more complicated when writes span object boundaries. Although the prototype does not currently address this situation, we are considering a number of potential solutions.

[2] We are investigating more scalable map distribution techniques that take advantage of the version tags on OSD requests and existing OSD chatter.

This work was performed under the auspices of the U. S. Department of Energy by University of California, Lawrence Livermore National Laboratory under contract W-7405-Eng-48.